CHAPTER 4

Enhancing the Desktop Interface

Application development normally begins with the creation of a simple version of the end product. There are certain things you must do to create a functional application that are included in this simple version. For example, you have to add business logic to process any data you've requested from the user. We discussed these skeleton applications in Chapter 3.

Once you have a skeleton application created and are fairly certain that it works as anticipated, you may want to turn your attention to bells and whistles and make the application more flexible or easier to use. For example, a skeleton application might include a File menu that contains an Exit command that allows you to close the application. An improvement to this menuing system might be a command that enables you to change the appearance of the display or to modify the behavior of the application itself.

Not all applications require every kind of enhancement that Visual C++ allows you to add. For example, adding menus to a dialog-based application may actually increase complexity and make it harder to use. Enhancements need to perform a basic service, make the application easier to use, or embellish the application in a way that doesn't impede the user, yet improves the application's aesthetic value.

Obviously, most of the enhancements that Visual C++ allows you to add to an application are visual. Menus and graphics both fall into this category. Other enhancements directly affect the way the user interacts with the application, like keyboard accelerators. Finally, some enhancements will remain unseen by the user until needed, like dialogs and error trapping.

In This Chapter

Enhancing the application flexibility with menus, keyboard accelerators, and toolbars

Using icons to improve the appearance of your application

Adding constant strings to the string table

Providing the user with version information about your application

Invoking events using timers

Improving application robustness with error trapping

Communicating with the user through dialogs

Writing applications for the handicapped using Windows accessibility features

Most of the resources that Visual C++ provides can be considered application enhancements. To see what kind of resources Visual C++ provides, right-click the <Project> Resources folder found on the ResourceView tab of the Workspace window and choose the Insert... command. You'll see an Insert Resource dialog box that contains a list of the resources that Visual C++ provides. Every project that Visual C++ can create also includes some default resources. It pays to know what these default resources are so that you can modify them as needed instead of creating new resources.

This chapter discusses a variety of application enhancements you can use for just about every application. Whether a particular enhancement is appropriate for your application depends on the value of that enhancement to the user. We'll also discuss ways that you can ensure an enhancement is valuable as the chapter progresses. By the time you finish this chapter, you should have a good idea of which enhancements to use to achieve a desired affect, which in turn will make your applications both more flexible and easier to use.

Menus

SDI and MDI applications normally include a default menu. Dialog applications, on the other hand, don't include a menu because they're normally used for small, utility applications. In both cases, you can add a new menu or embellish an existing one to create new functionality. In this section, we'll look at both menu strategies. First, we'll modify the menu for the SDI application we created in Chapter 3. Second, we'll add a menu to the dialog-based application in Chapter 3.

Modifying a Menu

More than likely you'll need to modify the predefined menus that come with your application. The MFC AppWizard normally provides just enough functionality so you can quickly prototype the essential application features. Anything that you want to add in addition to these basic user-interface features will normally need some type of menu and/or toolbar entry.

In this section, we'll look at what you need to do to modify a menu. In fact, I'll show you how to link the menu to a function that will activate it as well. Be sure you also take time to look at the next section that describes modifying the toolbar. There are ways to link a menu and toolbar selection using a single function so that the same thing will happen no matter which one the user clicks. Using a single function also ensures that the menu and toolbar remain in sync.

CD-ROM Location: Chap04\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDI.RC and MainFrm.CPP

The first thing you'll need to do is open the SDI application we created in the previous chapter. (I've copied this program to a new directory so that you'll be able to keep the Chapter 3 examples separate from those in this chapter.) The following procedure will show you how to modify the menu for our example — this same procedure will work for just about any menu modification you might need to make.

Try This

1. Click the Resource tab of the Workspace window. The entries on this tab will allow you to work with the resources that Visual C++ provides.
2. Open the SDI Resources | Menu folder, then double-click the IDR_MAINFRAME menu entry. You'll see the default application menu shown in Figure 4-1. This is the menu that the MFC AppWizard generates automatically. As you can see, it includes all of the default entries you need such as a File menu. Notice that there's a blank square right after the Help menu entry; this is the position of the next new menu entry. You'll also find these new menu item boxes below each of the main menu entries and on the side of submenu entries.

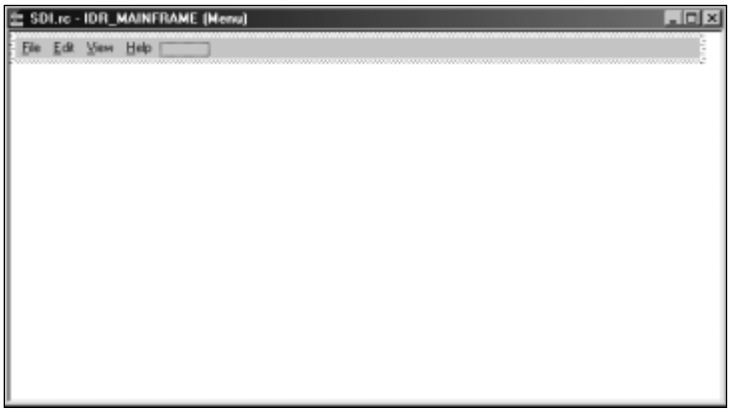

Figure 4-1: The default menu provided by the MFC AppWizard provides most of the general application features you would need.

3. Double-click the blank square next to the Help menu entry. You'll see the Menu Item Properties dialog box show in Figure 4-2.
4. Type **&My Menu** in the Caption field, then close the Menu Item Properties dialog. You'll see the menu entry added to the IDR_MAINFRAME menu.
5. Double-click the empty menu entry below the My Menu entry. You'll see the Menu Item Properties dialog. Type **ID_MY_MENU_OPTION_1** in the ID field and **Option &1** in the Caption field. Notice that the Prompt field is available in this case.

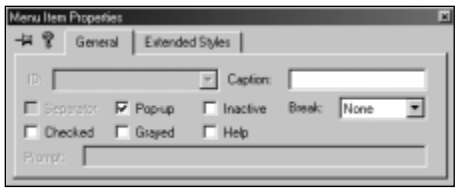

Figure 4-2: The Menu Item Properties dialog box allows you to define a new menu entry.

The Prompt field is divided into two parts separated by a new line escape sequence (\n). The first part gets displayed on the status line of the application. The second part appears as balloon help when the user passes their cursor over the top of the menu entry. This balloon help will only appear when a toolbar button with the same ID appears in the application.

6. Type **This is the first My Menu option.\nFirst My Menu Option** in the Prompt field. If you were to run the application right now, the status area would show "This is the first My Menu option." when you selected the My Menu | Option 1 entry. At this point we need to add a command handler so that the menu option can do something when selected by the user.

7. Ctrl+Double-click the My Menu | Option 1 entry. You'll see the MFC ClassWizard dialog box display shown in Figure 4-3. Notice that Visual C++ automatically selects the correct tab of the MFC ClassWizard for you, along with the correct Project, Class Name, and Object IDs field entries. All you need to do is choose the action that you want to handle.

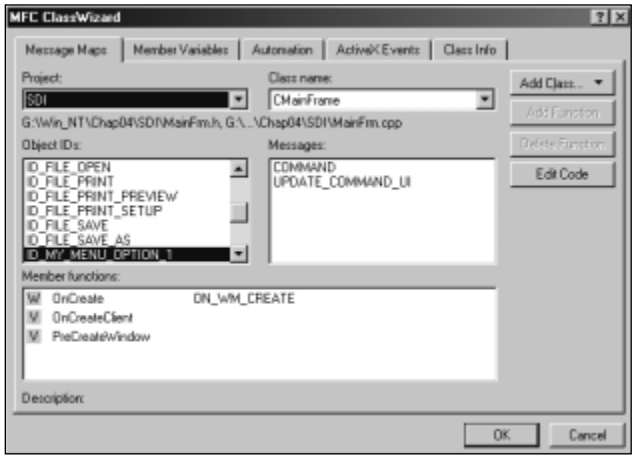

Figure 4-3: Visual C++ makes many of the selections for you automatically if you access the MFC ClassWizard using keyboard and mouse shortcuts.

8. Highlight COMMAND in the Messages field, then click Add Function. You'll see an Add Member Function dialog box like the one shown in Figure 4-4. This is where you'll decide on the name of the function that your application will use to handle any user input. Normally the default name works fine.

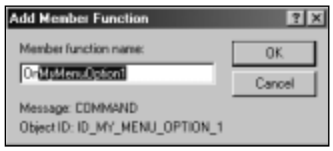

Figure 4-4: The Add Member Function dialog box is where you'll decide on the command handler function name.

Visual C++ normally uses default command handler function names that reflect the menu IDs you use. For this reason, it's normally a good idea to use a name hierarchy that reflects a menu's true position in the application. In this case, we used ID_MY_MENU_OPTION_1, which shows the hierarchy. As a result, all of the function handlers for the My Menu option will appear at the same location in the Workspace window.

9. Click OK to accept the default function name. Visual C++ will add a new function to your application. This function will allow you to monitor the user's selections from a central location.
10. Click Edit Code. Visual C++ will display the newly added function for you.

At this point, our new menu option is ready for some code. For the purposes of this example, we won't do anything too difficult. The whole purpose of this section is to get you started with menus — we'll explore more significant coding examples as the book progresses. Listing 4-1 shows the code you'll need to add to the OnMyMenuOption1( ) function.

Listing 4-1: **Simple Menu Addition Source Code**

```
void CMainFrame::OnMyMenuOption1()
{
    // Display a dialog box.
    AfxMessageBox("You chose Option 1 of the My Menu menu.",
        MB_OK | MB_ICONINFORMATION);

}
```

As you can see, the only thing that this simple function does is display a dialog box when the user selects the My Menu | Option 1 command from the application menu. Notice that I used the AfxMessageBox( ) function to simplify the code (it doesn't require arguments such as a title bar name). Also notice that most message boxes require both a command button constant (MB_OK) that affects the number and type of buttons the user sees and an icon constant (MB_ICONINFORMATION) that affects the icon displayed in the upper-left corner of the message box. Figure 4-5 shows what users will see when they execute the command.

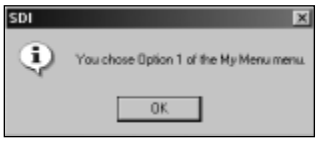

Figure 4-5: The only thing that our menu addition does is display this simple message box.

Creating a New Menu

There are going to be times when it's much easier to create the menus you need for a given situation during design time rather than attempt to modify the menu dynamically to meet a specific need. In fact, loading a predefined menu instead of modifying an existing menu will normally enhance application performance, albeit very slightly. The main reason to use multiple menus is to enhance developer productivity—it's easier to design a new menu than write the code to modify an existing one. You can see examples of this practice in the MDI example of Chapter 3. In that example, the MFC AppWizard automatically generated five different menus for you and loads them according to the application needs of the moment. All of this is done automatically for you, in this case, because the MFC AppWizard can predict the need for these menus.

Designing a menu is only the first step in the process of using it. You also need to write a small amount of code to load the menu as needed. In most cases, you'll simply define a member variable (if one doesn't already exist) and use it to load the menu as part of the window construction process. As an alternative, you can load the menu after the window is onscreen. We'll see examples of both menu-loading techniques in this section. The first menu that gets displayed during application initialization will allow you to display the second menu.

Once you have the menu displayed, most of the same practices that we explored in the previous section will come into play. You need to define some new menu entries and add the code required to handle any user interaction with them. However, in the previous example, we explored the case of a new custom menu entry. Microsoft provided automatic handlers for some common menu entries, making it unnecessary for you to write a separate handler for them. For example, the File | Exit menu command falls into this category—you can add the command to a new menu and the command will work fine, even if you don't write a custom handler for it.

"Technical note, TN022: Standard Commands Implementation" explains all of the standard commands that get implemented for you automatically. You can find this technical note in your Microsoft Developer Network (MSDN) Library (included with Visual Studio) or on the MSDN site online at http://msdn.microsoft.com/library/devprods/vs6/visualc/vcmfc/_mfcnotes_tn022.htm.

CD-ROM Location: Chap04\Dialog
Program Name: Dialog.EXE
Source Code Modules in Text: Dialog.RC and Dialog.CPP

Creating the Menu Resources

The first task is to create the menu. I'll assume that you already have the Dialog example from the previous chapter loaded. The following procedure will get you started.

Try This

1. Select the ResourceView tab of the Workspace window.
2. Right-click the Dialog Resources folder, and then choose Insert... from the context menu. You'll see the Insert Resource dialog box shown in Figure 4-6. Notice that this dialog box contains one entry for each type of resource and that there are subcategories of some resources like dialog boxes. We'll explore some of these subcategories as the book progresses.

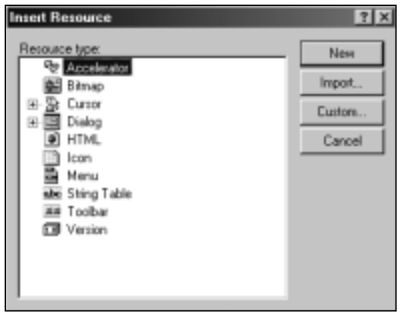

Figure 4-6: The Insert Resource dialog box allows you to add new resources to an existing project.

3. Highlight the Menu entry, then click New. Visual C++ will add a new menu to the list of resources in this project. Notice that it automatically adds a Menu folder to hold the new resource. The menu used to hold a resource category never appears until after you add at least one of that resource type to the project.

4. Right-click IDR_MENU1 in the Menu folder, then choose Properties from the context menu. You'll see the Menu Properties dialog box shown in Figure 4-7. Notice that this dialog box contains a Preview pane that you can use to view the menu without actually opening it.

Figure 4-7: The Menu Properties dialog box allows you to change the menu ID and view the menu's appearance without opening it.

5. Type **IDR_MAINFRAME** in the ID field. This will be our main menu.
6. Follows Steps 2 through 5 again to add the second menu. Give this menu an ID of IDR_SECONDARY.

Now that we have the two new menus created, it's time to define their appearance. Figure 4-8 shows what the IDR_MAINFRAME menu will look like. Use ID_MENU_CHANGE for the second menu entry ID. I also added a prompt to ID_MENU_CHANGE of "Select the secondary menu.\nChange Menus." You'll also need to add a member function for ID_MENU_CHANGE, just as we did in the previous section for the SDI example. I'll show you the code for this member function a little later.

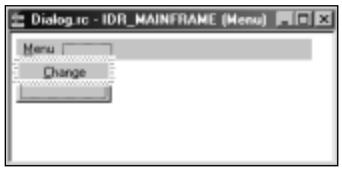

Figure 4-8: The only purpose for the first menu is to display the second menu.

Visual C++ will ask what class you want to associate IDR_MAINFRAME with when you create the menu handler. Specify the CDialogDlg class since this is the class we'll use for displaying the menus.

The second menu will look a bit more like the menus that you're used to seeing. It appears in Figure 4-9. Make absolutely certain that you give the second menu entry an ID of ID_APP_EXIT to ensure that the menu will behave as anticipated. We won't need to assign any code to this menu entry because it's handled automatically. I also gave ID_APP_EXIT a prompt of "Exit the program.\nQuit." Make certain that you don't add a member function for any part of this menu.

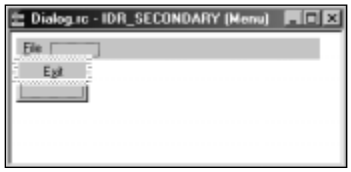

Figure 4-9: The secondary menu will allow us to exit the dialog application.

Making the Menus Functional

At this point, you should have two new menus and a menu command handler named OnMenuChange( ) added to your Dialog application. All we need is some code to make these two menus functional.

Listing 4-2: **Displaying a Menu During Dialog Startup Source Code**

```
CMenu* pSysMenu = GetSystemMenu(FALSE);
if (pSysMenu != NULL)
{
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())
    {
        pSysMenu->AppendMenu(MF_SEPARATOR);
        pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
    }
}

// Add a menu to the dialog box.
CMenu*    pMenu = new CMenu;
pMenu->LoadMenu(IDR_MAINFRAME);
SetMenu(pMenu);

// Set the icon for this dialog.  The framework does this automatically
//  when the application's main window is not a dialog
SetIcon(m_hIcon, TRUE);            // Set big icon
SetIcon(m_hIcon, FALSE);        // Set small icon
```

The first task is to display the IDR_MAINFRAME menu when the application first starts. This part of the application gets handled in different places depending on what application type you're working with. For our Dialog application, the OnInitDialog( ) function in the CDialogDlg class handles this task, so that's where we'll have to add the first bit of code. Listing 4-2 shows the code you'll need to add in bold (the surrounding code is shown for reference purposes).

As you can see, the addition required is relatively small and easy to understand. First, we create a CMenu object and initialize it. Next, the pMenu object gets loaded with the IDR_MAINFRAME menu. Finally, the SetMenu( ) method allows us to assign the menu object to the dialog box.

Creating a handler for loading the secondary menu is relatively easy as well. Listing 4-3 shows the OnMenuChange( ) function that you should have created as a handler for the Menu | Change command. Notice that this code is precisely the same as the code for initially loading the main menu. The only real difference is which menu is being loaded.

Listing 4-2: **Displaying the Secondary Menu Source Code**

```
void CDialogDlg::OnMenuChange()
{
    // Display the secondary menu.
    CMenu*    pMenu = new CMenu;
    pMenu->LoadMenu(IDR_SECONDARY);
    SetMenu(pMenu);

}
```

If you run the application at this point, you'll see the new menu added to the Dialog application as shown in Figure 4-10. (The IDD_DIALOG_DIALOG resource was modified slightly to make room for the menu in this figure — you'd need to make a similar change on a production application.) Choosing the Menu | Change command will display the secondary menu, which contains the File | Exit command. If you click the Exit entry, the application will stop execution, just as it does with the SDI application. Of course, the point here is that you didn't write any code to end the application — this feature is taken care of automatically for you because of the ID of the Exit entry.

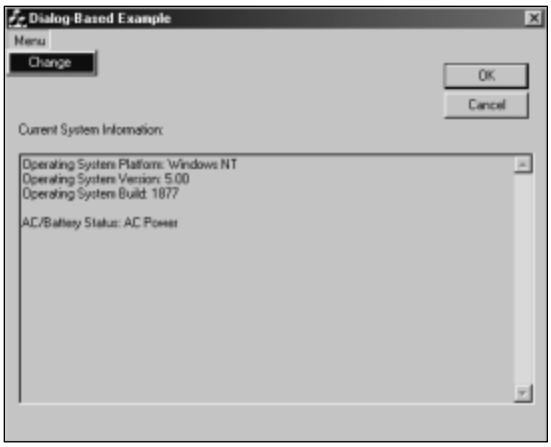

Figure 4-10: The updated Dialog application includes a new menu.

Keyboard Accelerators

Some people like to keep their hands on the keyboard at all times. To some extent, this really isn't possible anymore with Windows unless the user is willing to use a lot of extra keystrokes to get the job done. However, there are situations when a command that the user will access a lot is actually easier to use if you add some type of keyboard shortcut to the application. That's what keyboard accelerators are all about. They allow you to create keyboard shortcuts that keep the user's hands on the keyboard longer. This feature makes the user more efficient and improves productivity when used correctly.

There are many schools of thought when it comes to keyboard accelerators, as witnessed by their use in various applications installed on your machine right now. In some cases, the efficiency of using a keyboard accelerator is undeniable. For example, the Edit | Insert command appears on a menu, the toolbar, and as a keyboard accelerator in most applications. However, other keyboard accelerators aren't quite so clear cut. For example, many applications include a keyboard shortcut for the File | Exit (or Quit) command simply because it's a common application feature. It's important to differentiate between accelerators that you include in an application for productivity purposes and one that appears simply because the user may expect it. Filling your application with needless keyboard accelerators is one sure way to confuse the user and actually make this feature less useful than it could be.

The MFC AppWizard creates standard keyboard accelerators for you as part of creating the application shell. You can look at the menu that comes with the application and the accelerator resource to see which keyboard accelerators are provided by default. Figure 4-11 shows the keyboard accelerators provided with the SDI application we created in Chapter 3. (I've copied this application to a new directory so that you can see the changes made to it in Chapter 4.) As you can see, this list includes a bare minimum of accelerators that just about any application could use.

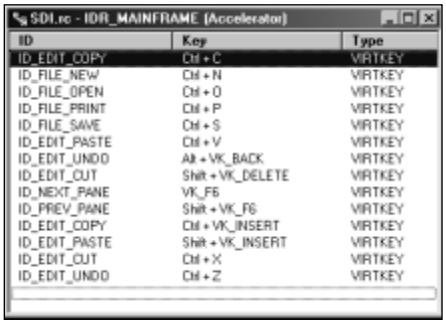
SDI.rc - IDR_MAINFRAME (Accelerator)
ID Key Type
ID_EDIT_COPY Ctrl + C VIRTKEY
ID_FILE_NEW Ctrl + N VIRTKEY
ID_FILE_OPEN Ctrl + O VIRTKEY
ID_FILE_PRINT Ctrl + P VIRTKEY
ID_FILE_SAVE Ctrl + S VIRTKEY
ID_EDIT_PASTE Ctrl + V VIRTKEY
ID_EDIT_UNDO Alt + VK_BACK VIRTKEY
ID_EDIT_CUT Shift + VK_DELETE VIRTKEY
ID_NEXT_PANE VK_F6 VIRTKEY
ID_PREV_PANE Shift + VK_F6 VIRTKEY
ID_EDIT_COPY Ctrl + VK_INSERT VIRTKEY
ID_EDIT_PASTE Shift + VK_INSERT VIRTKEY
ID_EDIT_CUT Ctrl + X VIRTKEY
ID_EDIT_UNDO Ctrl + Z VIRTKEY

Figure 4-11: The list of standard accelerators provided by the MFC AppWizard for an SDI application

Notice that there are three columns of information in Figure 4-11. The ID column associates the accelerator with a specific menu ID. In other words, the ID in this column has to match the one used for the menu entry. The second column contains the control-key combination that will execute the menu command. You can use any combination of the Ctrl, Alt, and Shift keys along with another keyboard character including all of the function and alphanumeric keys. Finally, you need to choose an accelerator type. The most common type for menus is a virtual key. You can also choose an ASCII key if so desired.

CD-ROM Location: Chap04\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDI.RC

One of the few menu commands that really needs a keyboard accelerator in a lot of cases is the File | Print Preview command. Many people like to keep the printed version of their document in mind as they type, which means they have to move their hands from the keyboard on occasion to use the toolbar or menu command. This section will look at adding a keyboard accelerator to the File | Print Preview command for the SDI application we looked at in Chapter 3. The following procedure will get you going (it is assumed that you already have the SDI application open in Visual C++).

Try This

1. Choose the ResourceView tab of the Workspace window.
2. Double-click on the IDR_MAINFRAME entry in the Accelerator folder. You'll see the accelerator table shown in Figure 4-11.
3. Double-click the empty box below the last accelerator table entry. You'll see the Accel Properties dialog shown in Figure 4-12. Notice that this dialog contains blanks for all of the column entries we saw in the accelerator table.

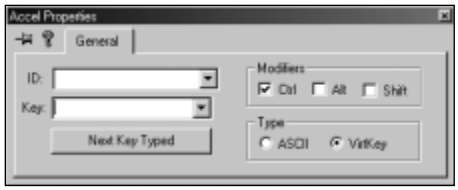

Figure 4-12: The Accel Properties dialog is where you associate a keyboard accelerator with a menu command.

4. Type **ID_FILE_PRINT_PREVIEW** in the ID field. This is the ID associated with the File | Print Preview menu command. (Note that you can also select this ID in the drop-down listbox.)
5. Click Next Key Typed, and then press Ctrl+Shift+P. This will make the appropriate entries in the dialog box for associating the Ctrl+Shift+P key combination with the File | Print Preview command.
6. Click the Close button. You should see the new keyboard accelerator added to the accelerator table. At this point, the keyboard accelerator is functional, but the user won't know it. We also need to add something to the menu so that the user knows which key combination to press to activate the File | Print Preview command from the keyboard.
7. Open the IDR_MAINFRAME entry found in the Menu folder.
8. Double-click the Print Preview entry in the File menu. You'll see a Menu Item Properties dialog (we looked at this dialog earlier in this chapter).
9. Type **"Print Pre&view\tCtrl+Shift+P"** in the Caption field. Notice that the Caption field uses a tab escape sequence to separate the menu name from the corresponding keyboard accelerator.
10. Close the Menu Item Properties dialog.

The application is ready to run. Simply compile the application and you'll find the accelerator is both documented and accessible to the user.

Toolbars

Just like keyboard accelerators speed menu access for keyboard lovers, toolbars are the productivity enhancement for mouse lovers. A toolbar button can save a user from having to navigate several layers of a menu hierarchy looking for a specific command. In addition, a toolbar button can act as a reminder that a specific feature exists. All the user needs to do is look across the toolbar to see a favorite feature represented as an icon.

In addition to buttons, a toolbar can also hold any other standard Windows form control like drop-down listboxes (which are the most common accessory). These additional features can help a user fine-tune the appearance of their display and generally make the application easier to use. You can also use drop-down listboxes for non-menu features like a style and font toolbar drop-down listboxes found with most word-processing applications.

Unfortunately, it's easy to abuse the functionality of a toolbar. I've seen some displays with five or six rows of buttons. When you reach this point, it may actually be easier to find what you need on the menu, rather than hunt through a plethora of buttons. Obviously, the complexity of the application and the knowledge of the user limit the number of buttons that an application can support. A user who knows how to use complex applications is more likely to make efficient use of a larger number of buttons.

Any Windows application can sport more than one toolbar. In fact, the SDI application in Chapter 3 includes two toolbars. (I've copied this application to a new directory to make the changes contained in this chapter.) The first contains the custom buttons for the Explorer interface used by the application (along with some standard buttons used by most applications on the market today like a print button). The second toolbar is blank. You use it to hold custom buttons.

Toolbars can actually be created from two kinds of resources. There's the traditional toolbar created using a toolbar resource and the dialog bar created using a standard dialog resource. The SDI application contains examples of both toolbar types and how to implement them.

In this section, we're going to look at a few interesting things you can do with toolbars. We'll begin by adding a toolbar for special commands. This will include adding some buttons for the new features we've added to the program. Next, we'll look at the dialog-based toolbar that appears below the main toolbar in the application. It's important to understand that, in many ways, this toolbar acts precisely the same as the traditional toolbar, but you also have some freedom in adding non-traditional controls.

CD-ROM Location: Chap04\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDI.RC and MainFrm.CPP

Adding a Toolbar

It's important to know how to add new toolbars to your application. Not only will some users want to access all of the features you've added to the application, but using separate toolbars for special features allows you (or the user) to hide non-essential features as well. For example, one user may use the File | Print Preview command all of the time, another may never use it.

Adding a new toolbar requires you to perform four tasks. First, you need to add the toolbar itself. Second, you need to add buttons to perform the required tasks. Third, you'll need to add code to add the toolbar to the application. Fourth, you'll need to add a menu entry and some code to control the appearance of the toolbar while the application is in operation. The following procedure will take you through all four tasks.

Try This

1. Right-click the Toolbar folder on the ResourceView tab of the Workspace window, and choose Insert Toolbar from the context menu. You'll see a new toolbar added to the application (Visual C++ should open it automatically for you).
2. Right-click the IDR_TOOLBAR1 entry and choose Properties from the context menu. You'll see the Toolbar Properties dialog show in Figure 4-13. This is where you'll change the ID and overall characteristics of the toolbar. Notice that this dialog also provides a preview of what the toolbar looks like so that you don't actually have to open the toolbar if it's closed.

Figure 4-13: The Toolbar Properties dialog allows you to see a preview of the toolbar and to change toolbar properties.

3. Type **IDR_SPECIAL** in the ID field, then close the Toolbar Properties dialog. Now it's time to add some buttons. Since the first button is already selected, we can draw an icon for the File | Print Preview command.
4. Draw an icon for the File | Print Preview command. Figure 4-14 shows the icon that I'll be using in this example.

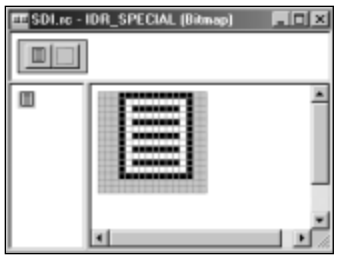

Figure 4-14: The IDR_SPECIAL toolbar will allow the user to select the File | Print Preview command.

5. Double-click the Print Preview button on the upper toolbar (not the drawing or the preview icon). You'll see a Toolbar Button Properties dialog like the one shown in Figure 4-15. This is where we'll associate the toolbar button with the File | Print Preview command.

Figure 4-15: The Toolbar Button Properties dialog allows you to associate the button with a menu command.

6. Choose ID_FILE_PRINT_PREVIEW in the ID field. Note that Visual C++ will automatically add a prompt for you after you close the Toolbar Button Properties dialog because there is already a prompt associated with this ID.
7. Close the Toolbar Button Properties dialog.

At this point, you have a fully functional toolbar. Now it's time to add the toolbar to our application. There's a very easy way to do this is by modifying the OnCreate( ) method of the CMainFrame class. Listing 4-3 shows the code you'll need to add in bold. The surrounding code shows you where you need to add the new code for this example.

Listing 4-3: **Adding a Toolbar Source Code**

```
if (!m_wndToolBar.CreateEx(this) ||
    !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
    TRACE0("Failed to create toolbar\n");
```

```
    return -1;      // fail to create
}

// Add our special toolbar.
if (!m_wndToolBar2.CreateEx(this) ||
    !m_wndToolBar2.LoadToolBar(IDR_SPECIAL))
{
    TRACE0("Failed to create special toolbar\n");
    return -1;      // fail to create
}

if (!m_wndDlgBar.Create(this, IDR_MAINFRAME,
    CBRS_ALIGN_TOP, AFX_IDW_DIALOGBAR))
{
    TRACE0("Failed to create dialogbar\n");
    return -1;        // fail to create
}

// Add our three toolbars to the Windows ReBar.
if (!m_wndReBar.Create(this) ||
    !m_wndReBar.AddBar(&m_wndToolBar) ||
    !m_wndReBar.AddBar(&m_wndDlgBar) ||
    !m_wndReBar.AddBar(&m_wndToolBar2))
{
    TRACE0("Failed to create rebar\n");
    return -1;      // fail to create
}

if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
{
    TRACE0("Failed to create status bar\n");
    return -1;      // fail to create
}

// Set both toolbars so they present tool tips.
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY);

m_wndToolBar2.SetBarStyle(m_wndToolBar2.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY);
```

As you can see, there are three basic steps to adding the toolbar to the application. First, create the toolbar object and load the special toolbar that we created earlier in the section. Second, because this application uses the Windows ReBar control, it's important to add the toolbar to the ReBar. Finally, it's important to set the toolbar to use tool tips so that the user knows precisely what the buttons on the toolbar are for.

We do need to add one more bit of code to this part of the example before you can compile it. The m_wndToolBar2 variable holds the second toolbar for our example. You need to declare it in the Mainframe.H file alongside the other window member variables as shown in the following code in bold.

```
protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;
    CReBar      m_wndReBar;
    CDialogBar      m_wndDlgBar;

    // Add a toolbar for special buttons.
    CToolBar    m_wndToolBar2;
```

Now we've come to the fourth task, adding some kind of menu functionality for displaying and removing the toolbar while the application is in operation. Obviously, we'll need to add a menu entry for this task. Open the IDR_MAINFRAME menu, then click on the View menu. Add a Special Commands entry to the bottom of the list of entries for this menu. Give this menu entry an ID of ID_VIEW_SPECIAL_COMMANDS and a prompt of "Display the special command toolbar.\nSpecial Command Toolbar". Since we'll display the toolbar as a default, you'll also want to select the Checked property for the new menu entry. You'll need to add a command handler as well, just as we did in the menu section of this chapter. Listing 4-4 shows the command handler code for displaying and hiding the toolbar.

Listing 4-4: **Displaying or Hiding a Toolbar Source Code**

```
void CMainFrame::OnViewSpecialCommands()
{
    CMenu*    pMenu = GetMenu();    // The menu for the application.

    // Determine if the bar is hidden or displayed.
    if (m_wndToolBar2.IsWindowVisible())
    {

        // Hide the bar.
        m_wndToolBar2.ShowWindow(SW_HIDE);

        // Uncheck the menu.
        pMenu->CheckMenuItem(ID_VIEW_SPECIAL_COMMANDS, MF_UNCHECKED);
    }
    else
    {

        // Show the bar.
        m_wndToolBar2.ShowWindow(SW_SHOW);
```

```
        // Check the menu.
        pMenu->CheckMenuItem(ID_VIEW_SPECIAL_COMMANDS, MF_CHECKED);
    }
}
```

This code is relatively simple to decipher. First, it checks to see if the window is visible or not. If the window is visible, then the code calls ShowWindow( ) with the SW_HIDE constant, which hides the window. It then unchecks the View | Special Commands menu item using the CheckMenuItem( ) method. The procedure for restoring the toolbar and rechecking the View | Special Commands menu item is about the same — only the constants differ.

Working with a Dialog Bar

Dialog bars look somewhat like toolbars when displayed onscreen, but their level of functionality is much different. You can use controls with a dialog bar that you normally can't use with a toolbar. The following procedure will help you set up two checkboxes on the dialog bar for our application that will allow the user to turn the toolbars on and off as needed.

Try This

1. Open the IDR_MAINFRAME dialog found in the Dialog folder of ResourceView.
2. Add two checkboxes to the left side of the dialog, one after another, so that it looks like a toolbar with two checkboxes.
3. Configure the two checkboxes as shown in Table 4-1. The final result should look like the dialog bar shown in Figure 4-16.
4. Shorten the dialog bar so that it is only slightly longer than the area required by the two checkboxes.

Figure 4-16: The final dialog bar product looks like a toolbar with specialized text buttons.

Table 4-1
Dialog Bar Checkboxes Configuration

Checkbox	*Field*	*Value*
First	ID	ID_VIEW_TOOLBAR
	Caption	Toolbar
	Pushlike	Checked
Second	ID	ID_VIEW_SPECIAL_COMMANDS
	Caption	Special Commands
	Pushlike	Checked

Shortening the default dialog bar length to the size needed for displaying the buttons will give the user more flexibility in moving the toolbars around.

The ID_VIEW_TOOLBAR button is automatically linked to its counterpart on the menu. On the other hand, we'll need to make some code modifications for the ID_VIEW_SPECIAL_COMMANDS button. The first place you'll need to add code is to the end of the OnCreate( ) method of the CMainFrame class. Here's what the code addition would look like:

```
// Set the Special Commands button on the dialog bar.
m_wndDlgBar.CheckRadioButton(ID_VIEW_SPECIAL_COMMANDS,
    ID_VIEW_SPECIAL_COMMANDS,
    ID_VIEW_SPECIAL_COMMANDS);
```

You'll need to add the same code to the else section of the OnViewSpecialCommands( ) method that we looked at in Listing 4-4. The if section of that method will require the following code addition:

```
// Set the Special Commands button on the dialog bar.
m_wndDlgBar.CheckRadioButton(ID_VIEW_SPECIAL_COMMANDS,
    ID_VIEW_SPECIAL_COMMANDS,
    0);
```

Notice that in all cases we're using the CheckRadioButton( ) method of m_wndDlgBar. The three arguments choose a starting and ending range of buttons that we want to work with, and the particular button within that range to check. Providing a value of 0 for the third argument tells CheckRadioButton( ) to uncheck all of the buttons in the range.

At this point, you can compile and run the application. Clicking the buttons on the dialog bar will either hide or show the respective toolbar. The pushlike appearance of the two checkboxes allows the user to see what appears to be the familiar toolbar buttons, without any loss of functionality.

Icons

Icons are used for a lot of different purposes in Windows applications. In Chapter 3, we looked at how you can load and unload icons for application use (see Listing 3-3). In this section, we'll talk about two very special icons stored in the IDR_MAINFRAME icon resource that you need to change for the application as a whole. These icons are used within Explorer, within the About dialog, and in the upper-left corner of the application for the system menu. We'll use the SDI application created in Chapter 3 for this example, though I've copied the source to a new directory so that you can see the changes made to it in this chapter.

In most cases, the icons you use will come in a single size and there will likely be one icon within the file. However, there are some situations in which you need to store more than one size of an icon in the same file. The IDR_MAINFRAME icon falls into this category. There's one 16 x 16-pixel icon and a second 32 ×32-pixel icon in the one resource. The 16 × 16 icon is used in situations that require a small icon, like the list view of Explorer or for the system menu. The 32 × 32 icon is used in situations that require a larger icon, like the About dialog or the large icons view of Explorer.

CD-ROM Location: Chap04\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDI.RC

However, the use of icons for the application isn't quite as straightforward as you might think due to some settings in the Display Properties dialog box (right-click the Desktop and choose Properties from the context menu). The user can check

the Show icons in all possible colors option on the Effects tab, which means that Windows will now display a secondary set of high-color icons if your application provides them and if the user has the display set to a high enough color depth. Unfortunately, there isn't a direct correlation between the number of colors you choose for your icon and the color depth chosen by the user. For example, if you choose a 256-color icon, then the user will need to choose a color depth greater than that since Windows reserves some colors for its own personal use. We're going to see how this particular feature works in this section of the chapter. The following procedure will get you started.

Try This

1. Open the IDR_MAINFRAME icon in the Icon folder of ResourceView. You'll see the standard 32 × 32-pixel MFC icon shown in Figure 4-17. The Device drop-down listbox allows you to choose from one of the existing icons. In this case there are two standard icons provided with the application.

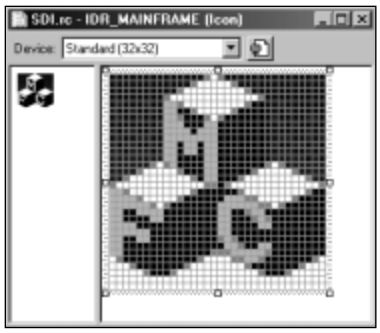

Figure 4-17: The IDR_MAINFRAME icon is actually composed of two images.

2. Click the New Device Image button (the one right next to the Device field in Figure 4-17). You'll see the New Icon Image dialog shown in Figure 4-18. Notice that there are some predefined icon types provided in the dialog that you can use, or you can create a custom icon.

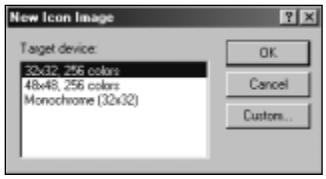

Figure 4-18: The New Icon Image dialog allows you to add images to an existing icon file.

3. Highlight the 32×32, 256 colors entry, then click OK. Visual C++ will add a new image to the icon file. Instead of the normal 16-color image though, this is a 256-color image that will only appear when the user has chosen the correct display settings.

4. Draw something on the image. Figure 4-19 shows the image that I created for this example. For a production application you would draw something closely associated with the application, but it doesn't really matter in this case.

The Visual C++ editor only contains 16 colors and won't allow you to create new ones. You can, however, draw the icon in another program like Paint, copy it to the clipboard, and then paste it to the image area. This will allow you to make use of the full 256-color capability offered by Visual C++ for icon images. Make sure that the image you create in an external graphics application is the same size as the icon you want to create in Visual C++, or the image won't fit in the space provided.

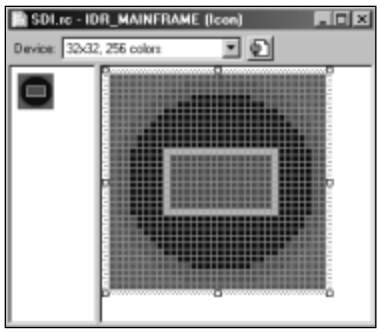

Figure 4-19: This is the image that you'll see when you set the Desktop and display settings for a high-color depth icon display.

5. Click the New Device Image button. You'll see the New Icon Image dialog shown in Figure 4-18.
6. Click Custom. You'll see the Custom Image dialog shown in Figure 4-20. This dialog allows you to create an image with custom settings. We need a small 256-color image to go with the standard image created in Step 3.

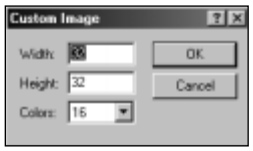

Figure 4-20: The Custom Image dialog allows you to create icons of a custom size and color depth.

Visual C++ won't warn you if you try to create an icon that is not a standard size. However, when you try to use the icon, the application will either ignore it or you'll see an error message. Icons should be either 16x16 pixels or 32x32 pixels in size (though other sizes may be acceptable in some situations).

7. Type **16** in the Width field and **16** in the Height field. Choose 256 in the Colors field. Click OK. You'll see a new entry in the Target Device list of the New Icon Image dialog. (The new entry should be highlighted for you.)
8. Click OK. Visual C++ will add the custom image to the icon file.
9. Draw something on the image. I drew an image similar to the one shown in Figure 4-19 for this example. At this point you should have four images in the IDR_MAINFRAME icon file as shown in Figure 4-21. Notice that the two 16-color images only list the image dimensions, not the color depth.

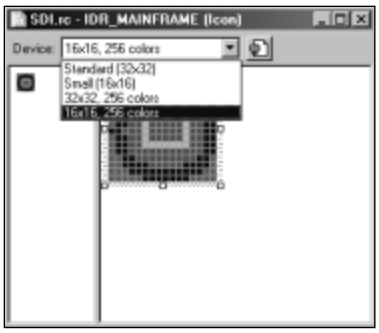

Figure 4-21: The IDR_MAINFRAME icon file now contains four images—each of which will get used in a different set of display conditions.

The two new icons are ready to go. Recompile your application (no new code is needed to use the icons; Windows will choose the appropriate icon automatically). Testing these two new icons is easy, but the point of the example is more far-reaching than the simple example might lead you to believe.

First, let's test the example. Display the program (found in the project Release folder) within Windows Explorer. Now, set your display to show at least 16-bit color. Once the display is set up, open the Effects tab of the Display Properties dialog. Uncheck the Show icons using all possible colors option, then click Apply. You should see the 16-color MFC icon displayed for the application in Explorer. Now, check the Show icons using all possible colors option, then click Apply. You should see the 256-color icon we just added to the application.

The whole purpose of this example is to show that you can create applications with a robust appearance that also scales to various user machine settings. We didn't have to add any special code or even detect the machine's state in this example. All that was required to support the higher color depth icon was the icon itself.

String and Version Tables

Most programmers know what strings are—they're simply text that gets displayed on screen or used in some other way. Visual C++ provides a wealth of ways to deal with strings and the textual content they contain. We've seen quite a few of those methods in this chapter.

Constants are one of the more efficient ways to store string values. You simply load the constant value from the application and display it onscreen. There's no requirement to create a variable and use up memory storing the string representation as a changeable value in memory.

Visual C++ automatically creates two resources for you that store constant string values as part of creating the application. The first is the String Table; the second is the version table (VS_VERSION_INFO).

The String Table contains all of the constant string resources that your application might need on a global basis. In fact, there are some values stored there automatically for you by Visual C++ like the menu prompts that we created earlier in the chapter. You'll also find values like the indicator displayed on the Status Bar when the user turns on Caps Lock. There isn't anything to stop you from creating other string constants manually and placing them here. For example, you might want to create constant strings that get displayed on the status bar or on the About dialog. Figure 4-22 shows an example of the String Table resource for the SDI application we created in Chapter 3.

There's a new version of this application in the Chapt04 directory so that you can see the changes made in this chapter.

Note that the ID field contains the value you use to load the string. In some cases, such as most message box function calls, you can use this ID value directly.

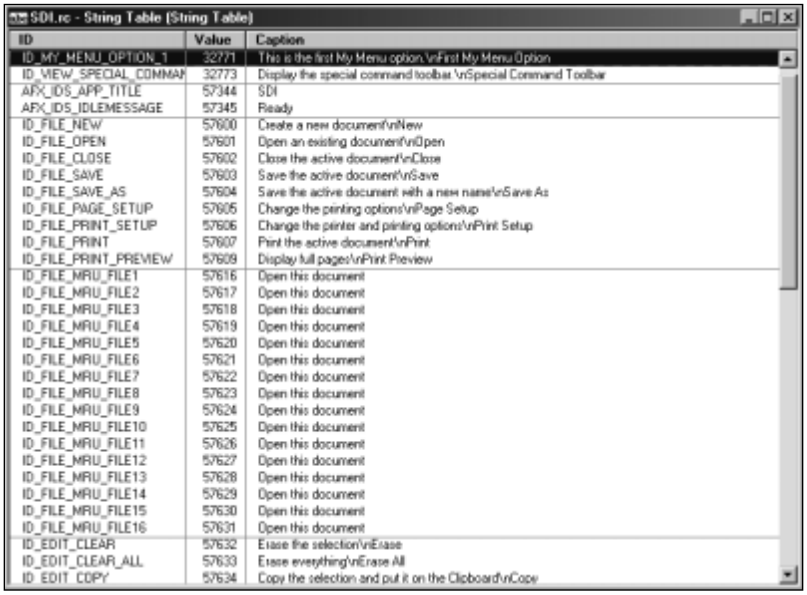

Figure 4-22: The String Table resource can contain a variety of constant strings, including many that are maintained automatically by Visual C++.

In many cases, the String Table represents the quickest way of making a group of changes to the global constant strings used in your application—in other cases it represents the only way of making a change. For example, you can't change the shift-key status indicators or the Title Bar text for your application anywhere else. While you can change the prompts for a menu by opening the individual property dialog, changing the String Table resource directly represents the fastest way of getting the job done when you have a group of changes to make.

Let's take a look at a simple way you can use a constant string in an application. Go to the bottom of the String Table and double-click on the blank line shown there. You'll see a String Properties dialog. Type **IDS_MENU_MESSAGE** in the ID field and **"You chose Option 1 of the My Menu menu."** in the Caption field. Close the String Properties dialog and you'll see the new string appear in the String Table. Now, let's modify the code for the MyMenuOption1( ) method of the CMainFrame class as shown in Listing 4-5.

CD-ROM Location: Chap04\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDI.RC and MainFrm.CPP

Listing 4-5: **Modified MyMenuOption1( ) Source Code**

```
void CMainFrame::OnMyMenuOption1()
{
    // Display a dialog box.
    AfxMessageBox(IDS_MENU_MESSAGE,
        MB_OK | MB_ICONINFORMATION);

}
```

Compile and run the application and you'll see that it runs as it did before. The only difference is that it uses a constant string. However, now that you have a constant string to use, you could repeat that dialog anywhere using precisely the same code. If you need to change the message, all you need to do is modify the constant string in the String Table, not all of the code instances in the application.

VS_VERSION_INFO is an important constant string resource for another reason. This is the information that someone will see when they right-click on your application and choose Properties from the context menu. The information in this string table appears on the Version tab and can help the user in a variety of ways. For one thing, it represents about the only efficient way to verify who created the file and what version of the file is in use (versus the version of the application as a whole). Figure 4-23 shows a typical unmodified VS_VERSION_INFO string table.

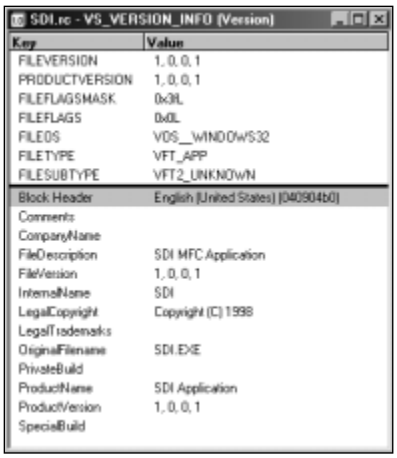

Figure 4-23: The VS_VERSION_INFO string table contains all of the version-specific information for your application.

As you can see, there are actually two sections to this string table. The version portion of the table identifies key elements of the file-versioning information. The FILEVERSION and PRODUCTVERSION entries work much as you might expect. They identify the version number of the file itself (the EXE in this case) and the product as a whole (which may not match the file version). One of the more interesting entries in this upper portion is FILEFLAGS. If you double-click this entry, you'll see a FILEFLAGS Properties dialog that contains one or more checkboxes. One of these checkboxes allows you to tell the user that this is a prerelease copy of the file — something that comes in quite handy during beta testing. You can also select the operating system type (there's a special setting of VOS_NT_WINDOWS32 for Windows NT exclusive files) and the file type/subtype. It's normally a good idea to leave these entries as is unless you have a good reason to change them manually.

The Block Header portion of VS_VERSION_INFO contains all of the string entries that Microsoft considers minimal for any application of the type that you're working with at the moment. These are the entries that appear on the Version tab of the <Application Name> Properties dialog. In other words, you should probably modify these entries to ensure that they are completely identified to anyone using it. Especially important is the copyright and trademark information. Users definitely need to know who owns the application and its components. This string represents the best way to mark the application itself in such a way that no confusion exists.

You can add additional Block Header entries to VS_VERSION_INFO by manually editing the <Application Name>.RC file using a program like Notepad. However, you must ensure that the Visual C++ IDE is not in use at the time, and that you use a text editor (not one that adds formatting information to the text).

Filling out the various version information blocks is as easy as typing the information you want to appear. You'll want to ensure that the VS_VERSION_INFO string table contains everything that you want it to by right-clicking the file in question (SDI.EXE in this case) and choosing Properties. Select the Version tab and you'll see something like the modified version information shown in Figure 4-24. Notice that this Version tab contains a special field named Development Status. I added it to the VS_VERSION_INFO string table using the code shown in Listing 4-6. (Part of the lower version information table is shown with the added text in bold.)

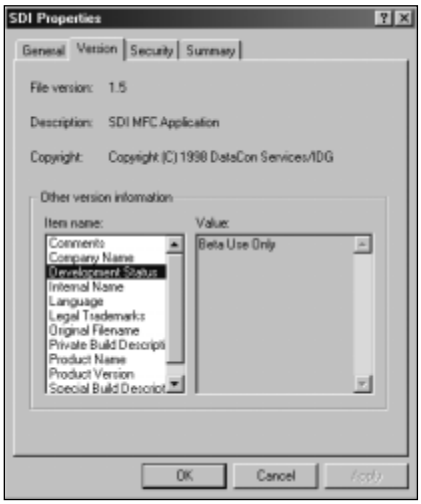

Figure 4-24: The Version tab of the SDI Properties dialog shows the version information for this application.

Listing 4-6: **Modified SDI.RC Source Code**

```
VALUE "LegalTrademarks", "\0"
VALUE "OriginalFilename", "SDI.EXE\0"
VALUE "PrivateBuild", "1.5\0"
VALUE "ProductName", "SDI Application\0"
VALUE "ProductVersion", "1.0\0"
VALUE "SpecialBuild", "\0"
VALUE "Development Status", "Beta Use Only\0"
```

Timers

Time: It's something that engulfs all of us. It's little wonder, then, that one of the more common event-driven additions to the desktop is the timer. The timer is the centerpiece of any time-related events on the computer. A timer can act as a clock, track the duration of an event, ensure that the user waits long enough (but not too long) for an action to take place, or cause a specific event to occur. In fact, timers are so important that they are included as standard components in most programming-language IDEs, including many of those provided by Microsoft. In addition, you'll find that Windows itself provides a variety of ways to access timers through the standard API calls. In other words, learning to create and use timers is an important part of learning to program within the Windows environment, and an important way to enhance the desktop application (as well as a variety of other application types).

This section will help you add basic timer functionality to the Dialog application we created in Chapter 3 in the form of a clock. (I've copied this application to a separate directory so that you can see the changes made in this chapter.) Obviously, there are more uses for the timer than a simple clock, but using a clock will allow you to see how a timer works more quickly.

CD-ROM Location: Chap04\Dialog
Program Name: Dialog.EXE
Source Code Modules in Text: Dialog.RC, DialogDlg.H, and DialogDlg.CPP

Creating a Clock Menu Entry

Just adding the clock would be relatively easy. However, it might be nice to allow the user to turn the clock on and off as well. After all, some users may not want to see a clock at the top of their application. With this in mind, we'll begin by adding a new menu item to the IDR_MAINFRAME menu. Create a menu structure like the one shown in Figure 4-25. Notice that this figure also contains the settings for the ID_CLOCK_START menu item.

Now that you have a new menu in place, you'll need to create a function for it (we explored this process earlier in the chapter). This menu is going to do something we haven't done in the past — it'll modify itself to show the current clock status. To do that, you'll need to create a global variable that you can declare in the DialogDlg.H file as shown in bold type in Listing 4-7. In addition to the menu-tracking variable, we'll also need a variable to hold the timer event-tracking variable. This variable also appears in Listing 4-7. Listing 4-8 shows the code for the OnClockStart( ) method (the method used to change the clock display status).

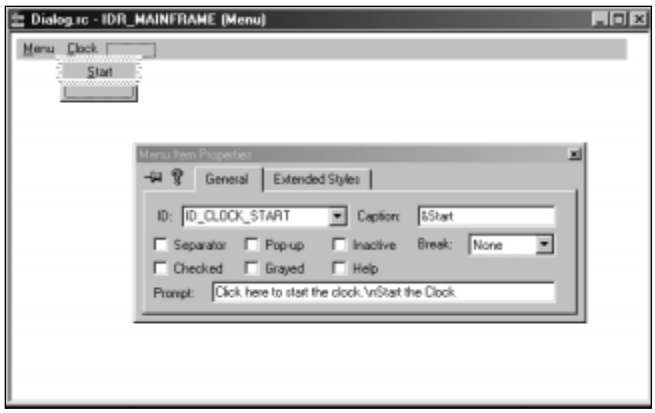

Figure 4-25: The Clock | Start command enables the user to start the clock.

Listing 4-7: **Variables Added to DialogDlg.H Source Code**

```
protected:
    HICON    m_hIcon;

    // Global clock status.
    BOOL    m_bClockOn;

    // Timer event tracking variable.
    UINT    m_nTimerID;
```

Listing 4-8: **OnClockStart() Method Source Code**

```
void CDialogDlg::OnClockStart()
{
    CMenu*    pMenu = GetMenu();

    if(m_bClockOn)
    {
        // Change the menu to reflect the next
        // clock status.
        pMenu->ModifyMenu(ID_CLOCK_START,
            MF_BYCOMMAND | MF_STRING,
            ID_CLOCK_START,
            "Start");
```

```
        // Set the clock status variable.
        m_bClockOn = false;

        // Hide the clock display.
        m_displayTime.ShowWindow(FALSE);

        // Destroy the timer.
        KillTimer(m_nTimerID);
    }
    else
    {
        // Change the menu to reflect the next
        // clock status.
        pMenu->ModifyMenu(ID_CLOCK_START,
            MF_BYCOMMAND | MF_STRING,
            ID_CLOCK_START,
            "Stop");

        // Set the clock status variable.
        m_bClockOn = true;

        // Show the clock display.
        m_displayTime.ShowWindow(TRUE);

        // Create a new timer with a 1 second upate.
        m_nTimerID = SetTimer(1, 1000, NULL);
    }

}
```

The OnClockStart( ) method begins by creating a variable to hold the menu. We'll need to modify the menu later, so it's important to perform this step at the beginning of the method code. The rest of the method is divided into two parts: one that handles the menu when the clock is stopped and another to handle the menu when the clock is started. Both parts operate about the same way. The first thing that happens is that the menu entry text gets changed. Notice that I use ID_CLOCK_START as both the existing and the new ID. You can assign different ID values for Stop and Start, but this will only serve to confuse the code and make it hard to read. Next, the code sets the clock status-tracking variable and either shows or hides the m_displayTime control. Finally, the code either sets or kills a timer, depending on whether you want to turn the clock on or off. This step is really important. Otherwise, you'll waste system resources by forcing Windows to handle an unnecessary timer when the clock is off.

Modifying the Dialog Display

Creating a place to put our clock on the dialog box comes next. Add an edit control to the dialog box for this application. Give it an ID of IDC_TIME_DISPLAY. You'll want to uncheck the Visible checkbox, set the Text Align property to Center, and check the Read-Only checkbox. Finally, size the control to 175 pixels. Your control should look like the one shown in Figure 4-26.

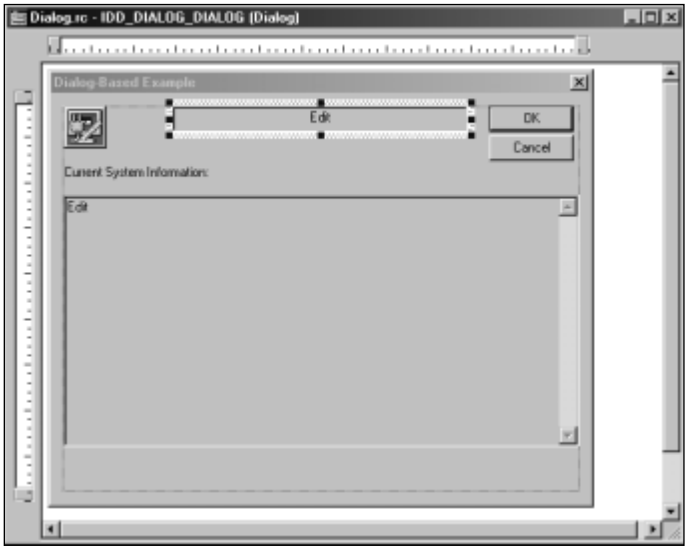

Figure 4-26: Our dialog box application now sports a new time-display edit control.

Access to this control requires the creation of a new variable. Ctrl+Double-click IDC_TIME_DISPLAY and you'll see an Add Member Variable dialog. Type m_displayTime in the Member variable name field and choose Control in the Category field. Click OK to create the new member variable.

Designing the Clock Handler

Now that we have all of the required support mechanisms in place, it's time to look at some code requirements. The first thing we'll need to do is create a function that will respond to the WM_TIMER message. The WM_TIMER message gets generated every time a timer expires--in other words, whenever it's time to do something in response to a timer event. Use the View | ClassWizard command to display the MFC ClassWizard dialog. Select CDialogDlg in the Class Name and Object IDs fields. Highlight the WM_TIMER entry in the Messages field. Click Add Function. Your MFC ClassWizard dialog should look like the one shown in Figure 4-27. Click Edit Code and you'll be taken to the new function. Listing 4-9 shows the code needed to add to this function.

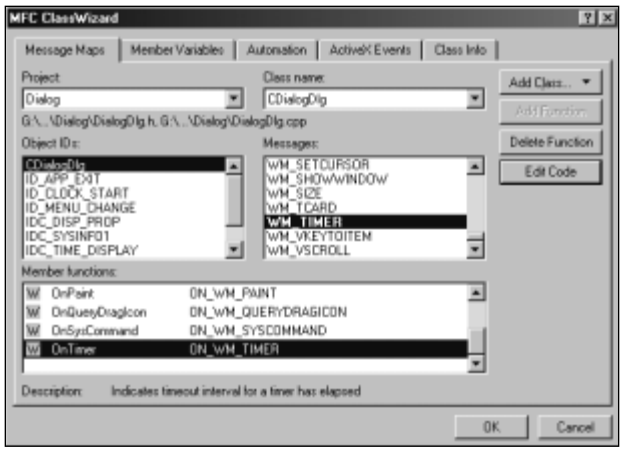

Figure 4-27: The place to begin adding a WM_TIMER message hander is MFC ClassWizard.

Listing 4-9: **OnTimer() Method Source Code**

```
void CDialogDlg::OnTimer(UINT nIDEvent)
{
    // Declare a series of variables to hold the time.
    COleDateTime    oDateTime;
    CString            sMonth;
    CString            sDay;
    CString            sYear;
    CString            sHour;
    CString            sMinute;
    CString            sSecond;
    CString            sDOW;

    int                iCurrentHour;        // Hour manipulation.
    LPTSTR            sBuffer = "0000";    // Date/Time buffer.
    CString            sAmPm = " a.m.";    // A.M./P.M. String.

    // Obtain the current time.
    oDateTime = COleDateTime::GetCurrentTime();

    // Get the day of week.
    switch (oDateTime.GetDayOfWeek())
    {
    case 1:
        sDOW = "Sunday";
        break;
    case 2:
        sDOW = "Monday";
        break;
    case 3:
```

Continued

Listing 4-9 *(continued)*

```
    sDOW = "Tuesday";
    break;
case 4:
    sDOW = "Wednesday";
    break;
case 5:
    sDOW = "Thursday";
    break;
case 6:
    sDOW = "Friday";
    break;
case 7:
    sDOW = "Saturday";
    break;
}

// Convert the date to text.
itoa(oDateTime.GetDay(), sBuffer, 10);
sDay = sBuffer;
if (sDay.GetLength() == 1)
    sDay = "0" + sDay;
itoa(oDateTime.GetMonth(), sBuffer, 10);
sMonth = sBuffer;
itoa(oDateTime.GetYear(), sBuffer, 10);
sYear = sBuffer;

// Convert the hour to am/pm format prior to
// coverting it to a string.
iCurrentHour = oDateTime.GetHour();
if (iCurrentHour > 12)
{
    iCurrentHour = iCurrentHour - 12;
    sAmPm = " p.m.";
}

// Convert the time to text.
itoa(iCurrentHour, sBuffer, 10);
sHour = sBuffer;
itoa(oDateTime.GetMinute(), sBuffer, 10);
sMinute = sBuffer;
if (sMinute.GetLength() == 1)
    sMinute = "0" + sMinute;
itoa(oDateTime.GetSecond(), sBuffer, 10);
sSecond = sBuffer;
if (sSecond.GetLength() == 1)
    sSecond = "0" + sSecond;

// Display the time.
```

```
    m_displayTime.SetWindowText(sDOW + " " +
        IntMonthToString(sMonth) + " " +
        sDay + ", " +
        sYear + " " +
        sHour + ":" +
        sMinute + ":" +
        sSecond);

    CDialog::OnTimer(nIDEvent);
}
```

The code for this function is relatively straightforward. The first task is to create come variables to hold all of the required time components. The second task is to obtain the current system time. Next, the code converts the COleDateTime object (oDateTime) into text components. The fourth task is to display the time onscreen using the m_displayTime variable that we created earlier. Finally, the OnTimer( ) method calls the default timer processing.

Adding an Error Trapping Function

There's a method call in Listing 4-9 that I didn't take time to explain before. It's the IntMonthToString( ) method used to convert the integer month passed by the system to our oDateTime object to a human readable string. Normally, you wouldn't separate this code into a separate method, but there are several good reasons to do so in this case. The following list provides some good ideas that you may want to use on your next application.

- **Code Reuse.** One of the more important reasons to separate this code is that you will probably need it for just about any kind of a time-related application need. It's very likely that the actual handling of time will vary from application to application, but the need for this utility method will not.
- **Code Readability.** Another good reason to separate this code from the main OnTimer( ) method is readability. By the time I reached the end of the OnTimer( ) method, the code was already getting long and cumbersome. Separating out the code that you could reuse in another application is a good way to clean up the appearance of individual methods within the current application as well.
- **Error Handling.** Finally, separating the code allowed me to perform a little error-trapping on the conversion process without making the OnTimer( ) method larger than it needed to be.

Listing 4-10 shows the IntMonthToString( ) method code. Make sure you also add an entry for it to the Protected section of the DialogDlg.H file.

Listing 4-10: **IntMonthToString( ) Method Source Code**

```
CString CDialogDlg::IntMonthToString(CString sMonth)
{
    int    iMonth = atoi(sMonth.GetBuffer(1));

    // Find the integer string value of sMonth and return
    // the month it corresponds to as a full text value.
    switch (iMonth)
    {
    case 1:
        return "January";
    case 2:
        return "February";
    case 3:
        return "March";
    case 4:
        return "April";
    case 5:
        return "May";
    case 6:
        return "June";
    case 7:
        return "July";
    case 8:
        return "August";
    case 9:
        return "September";
    case 10:
        return "October";
    case 11:
        return "November";
    case 12:
        return "December";
    }

    // Just in case we get some odd value from the calling
    // function.
    return "Error in Month Value: " + sMonth;
}
```

The code for the IntMonthToString( ) method is fairly easy to figure out. It begins by converting the month value to an integer (the oDateTime object returns it as an integer string). A simple switch structure converts the month number to a human readable string that gets returned to the calling function. If the value of sMonth is smaller than 1 or greater than 12, then we return an error message instead of a month value. Since the calling function is likely to display this string directly

onscreen, the user will see that there's a problem with the application and exactly what that problem is. We'll see other kinds of error handling as the book progresses, but this represents one of the easiest ways to handle a method-input error.

Accessibility Features

Recent versions of Windows have provided a new icon in the Control Panel, Accessibility. The features provided by this applet are designed to allow physically challenged users to make better use of the operating system in general, and specifically applications. For example, there's an Accessibility feature called MouseKeys that makes it easier for someone to move the mouse cursor by using the arrow keys instead of the mouse itself. There are other features for the hearing impaired like SoundSentry, and still others for those with visual problems like High Contrast.

Don't get the idea that the Accessibility features are only designed to help those with some type of physical challenge. You'll find that they come in quite handy for those of us who don't normally require them. For example, a full day of typing can tire anyone's eyes. Using the various Accessibility options for the sight-impaired can temporarily modify your display and make it easier to use when tired. (High Contrast is a personal favorite of mine.) It's important to keep this idea in mind when designing your application because many people won't think of using the Accessibility features if they don't normally need them. This means including easy-to-access menu or toolbar commands along with the appropriate help file entries.

This section of the chapter will show you how to activate and configure the various accessibility options. We'll add both visual and dexterity-oriented features to the SDI application we created in Chapter 3. These features will be accessible from the toolbar and within the menu to ensure that anyone who needs them will be able to find them with relative ease.

CD-ROM Location: Chap04\SDI
Program Name: SDI.EXE
Source Code Modules in Text: SDI.RC and MainFrm.CPP

Obviously, the first question we need to answer is what Accessibility features does Windows provide. In general, you'll use the GetSystemMetrics( ) function to get the current on or off status of a standard Windows Accessibility function. You'll use the SystemParamtersInfo( ) function to enable these functions or to check the current user settings for them. Table 4-1 contains a complete list of the Accessibility features, the access constant you use to access them, and a short description of what the Accessibility feature is used for.

Table 4-1
Windows Accessibility Features

Feature	*Get Status Constant*	*Set Status Constant*	*Description*
StickyKeys	SPI_ GETSTICKYKEYS	SPI_SETSTICKYKEYS	Makes the Ctrl, Alt, and Shift keys sticky, which means you can press them first and then the associated alphanumeric or function key. This allows users who can only press one key at a time to use complex keyboard accelerators.
FilterKeys	SPI_ GETFILTERKEYS	SPI_SETFILTERKEYS	Forces Windows to ignore brief or quickly repeated keystrokes. Also slows the repeat rate for the keyboard when a key is held down constantly.
ToggleKeys	SPI_ GETTOGGLEKEYS	SPI_SETTOGGLEKEYS	Sounds a tone whenever the Caps Lock, Num Lock, or Scroll Lock keys are pressed. You may need to disable this feature when an application requires extensive use of these keys, but remember to enable it again on program exit.
SoundSentry	SPI_ GETSOUNDSENTRY	SPI_SETSOUNDSENTRY	Displays a visual warning whenever the system makes a sound. You'll want to disable non-imperative sounds when this option is used.
ShowSounds	SPI_ GETSHOWSOUNDS	SPI_SETSHOWSOUNDS	Displays a caption whenever speech or sound occurs. See the SAMI sidebar for additional details on how closed captioning can enhance your application.
High Contrast	SPI_ GETHIGHCONTRAST	SPI_SETHIGHCONTRAST	This feature uses high- contrast colors and large fonts to make reading the screen easier. If your application has formatted displays (like those for database or dialog-based applications), you may need to adjust the display to make this feature useful.
MouseKeys	SPI_ GETMOUSEKEYS	SPI_SETMOUSEKEYS	Enables the user to use the numeric keypad keys in place of the mouse to move the mouse cursor onscreen. This feature shouldn't require changes to most applications. However, you may want to offer this feature when precise mouse placement is required, since using the cursor keys usually produces more accurate results.
SerialKey	SPI_ GETSERIALKEYS	SPI_SETSERIALKEYS	Allows alternative access to mouse and keyboard features. In general, you'll never need to directly access this feature in an application (unless the application requires a special input device).

The Microsoft Active Accessibility section of the Platform SDK reference contains complete guidelines for using the Accessibility features that appear in this section of the chapter. You'll want to spend some time looking at this section of the Platform SDK if you need to add special Accessibility features to your application that fall outside the range of those provided by Microsoft.

Designing the Support Menu and Toolbar

Our first order of business is to add a new menu to the application. Figure 4-28 shows the menu you'll need to design for this part of the SDI application. Table 4-2 contains a complete list of the menu item captions, their associated IDs, and prompts.

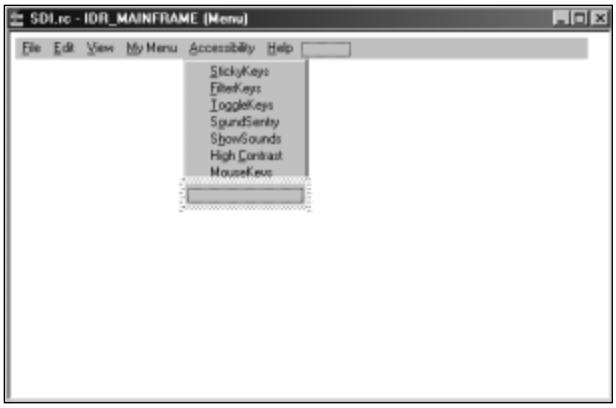

Figure 4-28: The Accessibility menu allows you to see the current Accessibility feature on/off state and change it.

We'll also need a toolbar containing the Accessibility feature options. This is another good place to use the dialog bar since we can show whether the accessibility feature is on or off. Use the same procedure as in the "Working with a Dialog Bar" section of this chapter to add some more buttons to the IDR_MAINFRAME dialog as shown in Figure 4-29. Make sure you use the same ID values for each button as we did for the associated menu entries. Notice that there's no requirement to keep the buttons on one line — a dialog bar is totally flexible with regard to layout.

Table 4-2
Accessibility Menu Entry Parameters

Caption	*ID*	*Prompt*
&StickyKeys	ID_ACCESSIBILITY_STICKYKEYS	Turn the StickyKeys Accessibility Feature On or Off\nStickyKeys
&FilterKeys	ID_ACCESSIBILITY_FILTERKEYS	Turn the FilterKeys Accessibility Feature On or Off\nFilterKeys
&ToggleKeys	ID_ACCESSIBILITY_TOGGLEKEYS	Turn the ToggleKeys Accessibility Feature On or Off\nToggleKeys
S&oundSentry	ID_ACCESSIBILITY_SOUNDSENTRY	Turn the SoundSentry Accessibility Feature On or Off\nSoundSentry
S&howSounds	ID_ACCESSIBILITY_SHOWSOUNDS	Turn the ShowSounds Accessibility Feature On or Off\nShowSounds
High &Constrast	ID_ACCESSIBILITY_HIGHCONTRAST	Turn the High Contrast Accessibility Feature On or Off\nHigh Contrast
&MouseKeys	ID_ACCESSIBILITY_MOUSEKEYS	Turn the MouseKeys Accessibility Feature On or Off\nMouseKeys

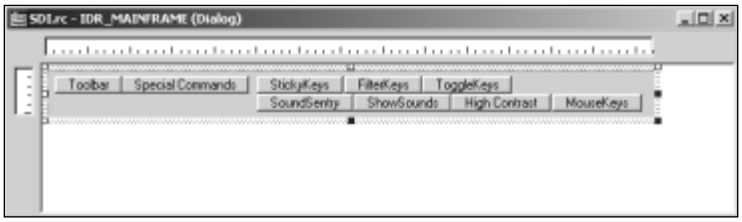

Figure 4-29: The dialog bar now contains a wealth of special command buttons.

Getting the Current Accessibility Feature Status

There are two reasons for getting the accessibility feature status. First, we need to update the menu and toolbar so that they reflect the user's current settings. Second, almost every one of the accessibility features relies on a data structure to hold various bits of information about their current status. ShowSounds is the exception in this. It's important to fill the required structures up as part of creating your application so that they're available when needed. So, the first thing we need to do is declare some accessibility structure variables in the protected area of the CMainFrame header as shown in Listing 4-11.

Listing 4-11: **Accessibility Feature Structures Source Code**

```
// Accessibility feature structures.
STICKYKEYS        m_structStickyKeys;
FILTERKEYS        m_structFilterKeys;
TOGGLEKEYS        m_structToggleKeys;
SOUNDSENTRY        m_structSoundSentry;
HIGHCONTRAST    m_structHighContrast;
MOUSEKEYS        m_structMouseKeys;
```

Now that we have some variables to use, let's look at the code used to fill the structures, and update the menus and toolbars. I created a new method for the CMainFrame class named SetAccessibilityStatus( ) and called it at the end of the OnCreate( ) method. You'll need to place the required method entry in the protected area of the MainFrm.H file. Listing 4-12 shows the source for the SetAccessibilityStatus( ) method.

Listing 4-12: **SetAccessibilityStatus( ) Method Source Code**

```
void CMainFrame::SetAccessibilityStatus()
{
    CMenu*    pMenu = GetMenu();
    BOOL    lShowSounds = GetSystemMetrics(SM_SHOWSOUNDS);

    // Determine if ShowSounds is set on.
    if(lShowSounds)
    {
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_SHOWSOUNDS,
            ID_ACCESSIBILITY_SHOWSOUNDS,
            ID_ACCESSIBILITY_SHOWSOUNDS);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_SHOWSOUNDS, MF_CHECKED);
    }

    // Get the current StickyKeys status.
    m_structStickyKeys.cbSize = sizeof(STICKYKEYS);
    if(!SystemParametersInfo(SPI_GETSTICKYKEYS,
        sizeof(STICKYKEYS),
        (LPVOID) &m_structStickyKeys,
        0))
        AfxMessageBox("GetStickyKeys Failed", MB_OK);

    // Set the menu and toolbar to reflect the status.
    if(m_structStickyKeys.dwFlags & SKF_STICKYKEYSON)
    {
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_STICKYKEYS,
            ID_ACCESSIBILITY_STICKYKEYS,
```

Continued

Listing 4-12 *(continued)*

```
        ID_ACCESSIBILITY_STICKYKEYS);

    pMenu->CheckMenuItem(ID_ACCESSIBILITY_STICKYKEYS, MF_CHECKED);
}

// Get the current FilterKeys status.
m_structFilterKeys.cbSize = sizeof(FILTERKEYS);
if(!SystemParametersInfo(SPI_GETFILTERKEYS,
    sizeof(FILTERKEYS),
    (LPVOID) &m_structFilterKeys,
    0))
    AfxMessageBox("GetFilterKeys Failed", MB_OK);

// Set the menu and toolbar to reflect the status.
if(m_structFilterKeys.dwFlags & FKF_FILTERKEYSON)
{
    m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_FILTERKEYS,
        ID_ACCESSIBILITY_FILTERKEYS,
        ID_ACCESSIBILITY_FILTERKEYS);

    pMenu->CheckMenuItem(ID_ACCESSIBILITY_FILTERKEYS, MF_CHECKED);
}

// Get the current ToggleKeys status.
m_structToggleKeys.cbSize = sizeof(TOGGLEKEYS);
if(!SystemParametersInfo(SPI_GETTOGGLEKEYS,
    sizeof(TOGGLEKEYS),
    (LPVOID) &m_structToggleKeys,
    0))
    AfxMessageBox("GetToggleKeys Failed", MB_OK);

// Set the menu and toolbar to reflect the status.
if(m_structToggleKeys.dwFlags & TKF_TOGGLEKEYSON)
{
    m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_TOGGLEKEYS,
        ID_ACCESSIBILITY_TOGGLEKEYS,
        ID_ACCESSIBILITY_TOGGLEKEYS);

    pMenu->CheckMenuItem(ID_ACCESSIBILITY_TOGGLEKEYS, MF_CHECKED);
}

// Get the current SoundSentry status.
m_structSoundSentry.cbSize = sizeof(SOUNDSENTRY);
if(!SystemParametersInfo(SPI_GETSOUNDSENTRY,
    sizeof(SOUNDSENTRY),
    (LPVOID) &m_structSoundSentry,
    0))
    AfxMessageBox("GetSoundSentry Failed", MB_OK);

// Set the menu and toolbar to reflect the status.
if(m_structSoundSentry.dwFlags & SSF_SOUNDSENTRYON)
```

```
    {
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_SOUNDSENTRY,
            ID_ACCESSIBILITY_SOUNDSENTRY,
            ID_ACCESSIBILITY_SOUNDSENTRY);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_SOUNDSENTRY, MF_CHECKED);
    }

    // Get the current High Contrast status.
    m_structHighContrast.cbSize = sizeof(HIGHCONTRAST);
    if(!SystemParametersInfo(SPI_GETHIGHCONTRAST,
        sizeof(HIGHCONTRAST),
        (LPVOID) &m_structHighContrast,
        0))
        AfxMessageBox("GetHighContrast Failed", MB_OK);

    // Set the menu and toolbar to reflect the status.
    if(m_structHighContrast.dwFlags & HCF_HIGHCONTRASTON)
    {
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_HIGHCONTRAST,
            ID_ACCESSIBILITY_HIGHCONTRAST,
            ID_ACCESSIBILITY_HIGHCONTRAST);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_HIGHCONTRAST, MF_CHECKED);
    }

    // Get the current MouseKeys status.
    m_structMouseKeys.cbSize = sizeof(MOUSEKEYS);
    if(!SystemParametersInfo(SPI_GETMOUSEKEYS,
        sizeof(MOUSEKEYS),
        (LPVOID) &m_structMouseKeys,
        0))
        AfxMessageBox("GetMouseKeys Failed", MB_OK);

    // Set the menu and toolbar to reflect the status.
    if(m_structMouseKeys.dwFlags & MKF_MOUSEKEYSON)
    {
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_MOUSEKEYS,
            ID_ACCESSIBILITY_MOUSEKEYS,
            ID_ACCESSIBILITY_MOUSEKEYS);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_MOUSEKEYS, MF_CHECKED);
    }
}
```

This may look like a lot of code, but there really isn't as much to getting the accessibility feature status as you might think. Let's begin with the easy case, ShowSounds. You can use a GetSystemMetrics( ) function call to determine the status of this particular Accessibility feature. In fact, that's the way Microsoft recommends that you determine it, even though you could use a SystemParametersInfo( ) call instead. If the function call returns true, then we'll check both the menu and the toolbar entry.

The other Accessibility features require a little more work, but it's the same work no matter which feature you want to use. The first thing you need to do is fill a structure of the appropriate type. For example, you'd fill a MOUSEKEYS structure for the MouseKeys feature. Make sure that you fill both the cbSize field of the structure and specify the size of the structure in the SystemParametersInfo( ) call as shown in the example code, or the call will fail.

Once you have a filled structure, you can check the dwFlags field for a particular flag value — in most cases it's something like MKF_MOUSEKEYSON. Notice that the first three letters are the letters for the Accessibility feature in question, followed by the feature name, and ending with the word "ON." If the comparison succeeds, then you need to check both the menu and toolbar buttons.

Setting the Accessibility Feature

Monitoring the status of the Accessibility features is nice, but being able to set them from the application is even better. This section will show you how to turn the Windows standard Accessibility features on and off with a push of a button. Obviously, making the Accessibility features this accessible will encourage users to use them. You'll need to create functions for each of the menu entries as we've done in the past. Listing 4-13 shows the code you need to make the Accessibility specific menu and toolbar entries for this example work.

Listing 4-13: Accessibility Feature On/Off Methods Source Code

```
void CMainFrame::OnAccessibilityStickykeys()
{
    CMenu*    pMenu = GetMenu();

    // Determine whether we need to set the StickyKeys
    // feature on or off.
    if(m_structStickyKeys.dwFlags & SKF_STICKYKEYSON)
    {

        // Set the StickyKeys status.
        m_structStickyKeys.dwFlags = m_structStickyKeys.dwFlags ^
            SKF_STICKYKEYSON;
        SystemParametersInfo(SPI_SETSTICKYKEYS,
            0,
            (LPVOID) &m_structStickyKeys,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_STICKYKEYS,
            ID_ACCESSIBILITY_STICKYKEYS,
            0);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_STICKYKEYS,
```

```
            MF_UNCHECKED);
    }
    else
    {

        // Set the StickyKeys status.
        m_structStickyKeys.dwFlags = m_structStickyKeys.dwFlags |
            SKF_STICKYKEYSON;
        SystemParametersInfo(SPI_SETSTICKYKEYS,
            0,
            (LPVOID) &m_structStickyKeys,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_STICKYKEYS,
            ID_ACCESSIBILITY_STICKYKEYS,
            ID_ACCESSIBILITY_STICKYKEYS);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_STICKYKEYS,
            MF_CHECKED);
    }
}

void CMainFrame::OnAccessibilityFilterkeys()
{
    CMenu*    pMenu = GetMenu();

    // Determine whether we need to set the FilterKeys
    // feature on or off.
    if(m_structFilterKeys.dwFlags & FKF_FILTERKEYSON)
    {

        // Set the FilterKeys status.
        m_structFilterKeys.dwFlags = m_structFilterKeys.dwFlags ^
            FKF_FILTERKEYSON;
        SystemParametersInfo(SPI_SETFILTERKEYS,
            0,
            (LPVOID) &m_structFilterKeys,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_FILTERKEYS,
            ID_ACCESSIBILITY_FILTERKEYS,
            0);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_FILTERKEYS,
            MF_UNCHECKED);
    }
    else
    {

        // Set the FilterKeys status.
```

Continued

Listing 4-13 *(continued)*

```
        m_structFilterKeys.dwFlags = m_structFilterKeys.dwFlags |
            FKF_FILTERKEYSON;
        SystemParametersInfo(SPI_SETFILTERKEYS,
            0,
            (LPVOID) &m_structFilterKeys,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_FILTERKEYS,
            ID_ACCESSIBILITY_FILTERKEYS,
            ID_ACCESSIBILITY_FILTERKEYS);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_FILTERKEYS,
            MF_CHECKED);
    }
}

void CMainFrame::OnAccessibilityTogglekeys()
{
    CMenu*    pMenu = GetMenu();

    // Determine whether we need to set the ToggleKeys
    // feature on or off.
    if(m_structToggleKeys.dwFlags & TKF_TOGGLEKEYSON)
    {

        // Set the ToggleKeys status.
        m_structToggleKeys.dwFlags = m_structToggleKeys.dwFlags ^
            TKF_TOGGLEKEYSON;
        SystemParametersInfo(SPI_SETTOGGLEKEYS,
            0,
            (LPVOID) &m_structToggleKeys,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_TOGGLEKEYS,
            ID_ACCESSIBILITY_TOGGLEKEYS,
            0);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_TOGGLEKEYS,
            MF_UNCHECKED);
    }
    else
    {

        // Set the ToggleKeys status.
        m_structToggleKeys.dwFlags = m_structToggleKeys.dwFlags |
            TKF_TOGGLEKEYSON;
        SystemParametersInfo(SPI_SETTOGGLEKEYS,
            0,
            (LPVOID) &m_structToggleKeys,
```

```
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_TOGGLEKEYS,
            ID_ACCESSIBILITY_TOGGLEKEYS,
            ID_ACCESSIBILITY_TOGGLEKEYS);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_TOGGLEKEYS,
            MF_CHECKED);
    }
}

void CMainFrame::OnAccessibilitySoundsentry()
{
    CMenu*    pMenu = GetMenu();

    // Determine whether we need to set the SoundSentry
    // feature on or off.
    if(m_structSoundSentry.dwFlags & SSF_SOUNDSENTRYON)
    {

        // Set the SoundSentry status.
        m_structSoundSentry.dwFlags = m_structSoundSentry.dwFlags ^
            SSF_SOUNDSENTRYON;
        SystemParametersInfo(SPI_SETSOUNDSENTRY,
            0,
            (LPVOID) &m_structSoundSentry,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_SOUNDSENTRY,
            ID_ACCESSIBILITY_SOUNDSENTRY,
            0);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_SOUNDSENTRY,
            MF_UNCHECKED);
    }
    else
    {

        // Set the SoundSentry status.
        m_structSoundSentry.dwFlags = m_structSoundSentry.dwFlags |
            SSF_SOUNDSENTRYON;
        SystemParametersInfo(SPI_SETSOUNDSENTRY,
            0,
            (LPVOID) &m_structSoundSentry,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_SOUNDSENTRY,
            ID_ACCESSIBILITY_SOUNDSENTRY,
            ID_ACCESSIBILITY_SOUNDSENTRY);
```

Continued

Listing 4-13 *(continued)*

```
        pMenu->CheckMenuItem(ID_ACCESSIBILITY_SOUNDSENTRY,
            MF_CHECKED);
    }
}

void CMainFrame::OnAccessibilityShowsounds()
{
    CMenu*    pMenu = GetMenu();

    // Determine whether we need to set the ShowSounds
    // feature on or off.
    if(GetSystemMetrics(SM_SHOWSOUNDS))
    {

        // Set the ShowSounds status.
        SystemParametersInfo(SPI_SETSHOWSOUNDS,
            FALSE,
            NULL,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_SHOWSOUNDS,
            ID_ACCESSIBILITY_SHOWSOUNDS,
            0);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_SHOWSOUNDS,
            MF_UNCHECKED);
    }
    else
    {

        // Set the ShowSounds status.
        SystemParametersInfo(SPI_SETSHOWSOUNDS,
            TRUE,
            NULL,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_SHOWSOUNDS,
            ID_ACCESSIBILITY_SHOWSOUNDS,
            ID_ACCESSIBILITY_SHOWSOUNDS);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_SHOWSOUNDS,
            MF_CHECKED);
    }
}

void CMainFrame::OnAccessibilityHighcontrast()
{
    CMenu*    pMenu = GetMenu();
```

```
    // Determine whether we need to set the High Contrast
    // feature on or off.
    if(m_structHighContrast.dwFlags & HCF_HIGHCONTRASTON)
    {

        // Set the High Contrast status.
        m_structHighContrast.dwFlags = m_structHighContrast.dwFlags ^
            HCF_HIGHCONTRASTON;
        SystemParametersInfo(SPI_SETHIGHCONTRAST,
            0,
            (LPVOID) &m_structHighContrast,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_HIGHCONTRAST,
            ID_ACCESSIBILITY_HIGHCONTRAST,
            0);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_HIGHCONTRAST,
            MF_UNCHECKED);
    }
    else
    {

        // Set the High Contrast status.
        m_structHighContrast.dwFlags = m_structHighContrast.dwFlags |
            HCF_HIGHCONTRASTON;
        SystemParametersInfo(SPI_SETHIGHCONTRAST,
            0,
            (LPVOID) &m_structHighContrast,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_HIGHCONTRAST,
            ID_ACCESSIBILITY_HIGHCONTRAST,
            ID_ACCESSIBILITY_HIGHCONTRAST);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_HIGHCONTRAST,
            MF_CHECKED);
    }
}

void CMainFrame::OnAccessibilityMousekeys()
{
    CMenu*    pMenu = GetMenu();

    // Determine whether we need to set the MouseKeys
    // feature on or off.
    if(m_structMouseKeys.dwFlags & MKF_MOUSEKEYSON)
    {

        // Set the MouseKeys status.
```

Continued

Listing 4-13 *(continued)*

```
        m_structMouseKeys.dwFlags = m_structMouseKeys.dwFlags ^
            MKF_MOUSEKEYSON;
        SystemParametersInfo(SPI_SETMOUSEKEYS,
            0,
            (LPVOID) &m_structMouseKeys,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_MOUSEKEYS,
            ID_ACCESSIBILITY_MOUSEKEYS,
            0);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_MOUSEKEYS,
            MF_UNCHECKED);
    }
    else
    {

        // Set the MouseKeys status.
        m_structMouseKeys.dwFlags = m_structMouseKeys.dwFlags |
            MKF_MOUSEKEYSON;
        SystemParametersInfo(SPI_SETMOUSEKEYS,
            0,
            (LPVOID) &m_structMouseKeys,
            0);

        // Set the menu and toolbar to reflect the status.
        m_wndDlgBar.CheckRadioButton(ID_ACCESSIBILITY_MOUSEKEYS,
            ID_ACCESSIBILITY_MOUSEKEYS,
            ID_ACCESSIBILITY_MOUSEKEYS);

        pMenu->CheckMenuItem(ID_ACCESSIBILITY_MOUSEKEYS,
            MF_CHECKED);
    }
}
```

Again, there's a lot of code here, but most of these functions fall into one of two categories. Let's discuss the easy one first, the OnAccessibilityShowsounds( ) method. This is the easiest of the Accessibility toggle methods because the ShowSounds Accessibility feature doesn't require any kind of a structure. All this code does is use the GetSystemMetrics( ) function to determine the current ShowSounds( ) state. Once the state is determined, the feature gets turned off or on using about the same set of steps. The first task is to use the SystemParametersInfo( ) function with the SPI_SETSHOWSOUNDS constant to change the feature state. Notice where the TRUE and FALSE values go—it's where we normally place the length of the data structure. Once the ShowSounds feature state is set, the code makes two calls to check the menu and toolbar entries.

Now let's look at the other Accessibility feature toggle method. I'll use the StickyKeys feature as the basis for this discussion, but all of the other functions work the same. As with the OnAccessibilityShowsounds( ) method, the OnAccessibilityStickykeys( ) method begins by determining the feature state. Once the feature state is determined, the method follows the same set of steps to turn it on or off. Notice that the first thing we need to do is change the dwFlags field of the m_structStickyKeys structure. The method uses an exclusive or to turn the flag off, and an inclusive or to turn it on. Notice that the call to SystemParametersInfo( ) doesn't require a structure size this time. That's because the cbSize field already contains this value. (Remember that the first thing this application does is fill the structures for you so that you don't need to fill them later when it's time to toggle the feature state.) Once the feature state is toggled, the method calls the appropriate function to change the menu and toolbar check status.

At this point, you have a fully functional application for testing the Accessibility features that Windows has to offer. Make sure that you've actually installed this feature by looking for the Accessibility icon in the Control Panel. Once you're sure that the Accessibility features are installed, you'll want to spend some time experimenting with this application to see what's possible using Windows native capabilities.

Closed Captioning: Not Just for Television Anymore

Closed captioning has moved from your television to the computer to assist those who are physically challenged. Closed captioning is an alternate form of audio content. It may include descriptions of sounds, symbols, icons, or text to represent the audio content.

The use of closed captioning in applications affects many people. The most obvious users are persons with hearing loss or impairment. It's also useful for people who are learning to read, learning a second language, or in situations in which noise is unwelcome, such as libraries or multi-use offices.

Microsoft's Synchronized Accessible Media Interchange (SAMI) improves delivery of closed captioning with multimedia applications by time-synchronizing the captioning file with the media file. This makes it easier to edit or change either file than it is in other applications where the information is encoded within the media file.

SAMI can provide closed captioning in more than one language and with different presentation possibilities. The user chooses the appearance of the captions by selecting the color, font, and size of the text. This will increase the ease of reading for children and for people with slight visual impairments, who can select larger screen types. The user also chooses which language the text will appear in, such as American English or Canadian French.

SAMI files use the extension .SMI or .SAMI. The file format specification is free (no licensing fee). You can find the specification, demonstrations, and examples at `http://www.microsoft.com/enable/`, which is Microsoft's Accessibility Web site. The demonstrations require Internet Explorer 4 (`http://www.microsoft.com/ie/`) and Windows Media Player (`http://www.microsoft.com/windows/mediaplayer/`).

Continued

(continued)

SAMI is constructed like, though not the same as, HTML. This makes it an easy format to learn. For example, documents begin with a <SAMI> tag and end with a </SAMI> tag, replacing the <HTML> tag in a normal HTML document. (The tags must be uppercase.) Once you have the <SAMI> tag in place, a SAMI document looks much the same as its HTML counterpart. It even includes the <HEAD> and <BODY> tags.

Summary

In this chapter, I have shown you many of the major desktop-application enhancements you can make — we haven't even begun to cover them all. However, there are some general conclusions you can draw from the things we did cover:

- ✦ Always check to see which of these enhancements are a must-have for the user, and which would be overkill and could actually slow the user down.
- ✦ Performance is another consideration. Every time you add a desktop enhancement, you have to be sure that the enhancement will result in some kind of tangible gain for the user.
- ✦ Menus aren't always restricted to traditional roles. Try using a menu with a dialog-based application that seems a little too cluttered. Multiple menus are handy when your application could use some extra flexibility and the user could use some clarification about what's possible in a given application state.
- ✦ There are two kinds of toolbars you should consider for every application: traditional and dialog bar.
- ✦ The traditional toolbar is the best option when you need to implement standard features using the fewest system resources.
- ✦ The dialog bar is the best option when you need maximum flexibility, or need to implement unusual toolbar control features.
- ✦ Many people think that you can only use two icons for an application. Icons can now provide different sizes and color depths, which means that you can have multiple application icons that get used according to the user's Windows setup.
- ✦ String tables can save you considerable typing when you need to use the same string constant for a message or other purpose.
- ✦ Version tables contain essential information about your application. Especially important are the copyright and trademark entries.

- You're not limited to the Version table entries that Microsoft provides as a default. However, changing the Version table entries does require manually editing the associated application RC file.
- Timers can help regulate the user's application experience in a lot of ways. Don't forget that they can be used for more than time keeping.
- Accessibility is for everyone. It doesn't matter whether you're physically challenged or not. There are times where everyone can use a little help to get that last ounce of productivity.

✦ ✦ ✦